A Consistency Checker for Memory Subsystem Traces

Matthew Naylor, Simon W. Moore, Alan Mujumdar
Computer Laboratory, University of Cambridge, UK
{matthew.naylor, simon.moore, alan.mujumdar}@cl.cam.ac.uk

***Abstract*—Verifying the memory subsystem in a modern shared-memory multiprocessor is a big challenge. Optimized implementations are highly sophisticated, yet must provide subtle consistency and liveness guarantees for the correct execution of concurrent programs. We present a tool that supports efficient specification-based testing of the memory subsystem against a range of formally specified consistency models. Our tool operates directly on the memory subsystem interface, promoting a compositional approach to system-on-chip verification, and can be used to search for simple failure cases – assisting rapid debug. It has recently been incorporated into the development flows of two open-source implementations – Berkeley's Rocket Chip (RISC-V) and Cambridge's BERI (MIPS) – where it has uncovered a number of serious bugs.**

I. Introduction

We are interested in verifying that the memory subsystem in a shared-memory multiprocessor implements a well-defined consistency model – a pre-requisite for the correct execution of concurrent programs on such architectures [1]. We take a *specification-based testing approach* inspired by the work of Manovit et al. [2], [3], [4], [5] and their *TSOtool* [6]. TSOtool generates pseudo-random multi-threaded programs, runs them on a multiprocessor, and compares the results against the *Total Store Order* specification (TSO) to reveal potential discrepancies. A key contribution of the work is a state-of-the-art conformance-checking algorithm for TSO that can handle long-running programs – on the order of millions of memory operations and hundreds of cores – *despite this being an NP-complete problem* [7]. TSOtool, and variants of it, have been used with great success at Sun Microsystems [2] and Intel [8].

In this paper, we both build on and deviate from TSOtool in a number of useful ways, as outlined below.

Testing the memory subsystem in isolation Unlike TSOtool, we feed memory requests directly to the memory subsystem using an HDL-level test bench, not via software running on processors connected to the memory subsystem. As a result:

- The memory subsystem can be tested as a reusable component, not constrained to the usage pattern of any particular processor implementation.
- Greater stress can be applied to the memory subsystem directly than may be possible indirectly via software.
- It is faster to simulate the memory subsystem in the absence of processor pipelines, allowing more tests per unit time.
- We avoid the implicit traffic arising from execution of software tests (e.g., fetching instructions, logging test results), allowing simpler failure cases to be found.

	Model	Name & Reference
	SC	Sequential Consistency [10]
$\subset$	TSO	Total Store Order [11]
$\subset$	PSO	Partial Store Order [11]
$\subset$	WMO[1]	Weak Memory Order [11]
$\subset$	POW	POWER model [12]

Fig. 1: Consistency models supported by Axe

More consistency models We can test memory subsystems against a range of consistency models found in modern multiprocessors, not just TSO. For example, we can test Berkeley's Rocket Chip [9], which at the time of writing is intentionally more relaxed than TSO.

Our conformance-checking tool – *Axe* – supports a spectrum of five consistency models shown in Figure 1, each one permitting a subset of the behaviors allowed by the next. In this paper, we focus on support for the SPARC models (SC, TSO, PSO, WMO), which have been sufficient for the memory subsystems we have tested thus far; support for the POWER model (POW) is detailed in the Axe manual [14]. Our checking algorithm for the SPARC models is a generalization of TSOtool's algorithm. Although this generalization leads to a checker with a worse time and space complexity, we show that it still performs very well in practice.

Simpler debugging The TSOtool authors say very little about how best to report violations to the user. Simply indicating that a violation exists is clearly not very helpful when large traces are involved. However, due to the backtracking nature of the checker, it may not be easy to give a concise error message. To address this, we have developed a shrinking procedure that attempts to isolate the smallest subset of a failing trace that still violates the model.

While this procedure works very well for explaining why the model has been violated, it does not always help in understanding what went wrong in the implementation. For this, we exploit one of the great benefits of specification-based testing: we adjust the test-generation method to search for small test-cases that fail.

Open-source tools While TSOTool is a *"proprietary product of Sun Microsystems"* [6], Axe is open-source and freely-available [14], as are the applications of Axe to open-source processors Rocket Chip and BERI [9], [15].

Paper outline We begin by presenting the design and implementation of Axe. This includes the format of memory subsys-

[1]WMO is equivalent to SPARC RMO [11] except that it forbids reordering of loads *to the same address*, making it a subset of POWER [12].

tem traces taken as input, the consistency models supported, the checking algorithms we have implemented, performance evaluations of these algorithms, and a tool for shrinking failing traces to reveal minimal violations. After that, we present experiences of using Axe to test the memory subsystem in Berkeley's open-source Rocket Chip [9], including details of test benches developed and the bugs we found. Finally, we compare our approach against litmus testing and with other checking tools reported in the literature. This includes experiences of testing the open-source BERI processor [15], the value of searching for small failure cases, and bugs missed by Axe.

II. Axe consistency checker

Given a *memory subsystem execution trace* containing a set of top-level memory requests and responses (including loads, stores, atomic read-modify-writes, memory barriers, and optional timestamps) initiated by concurrent processor cores (or "hardware threads"), Axe determines whether the trace is valid according to one of the consistency models listed in Figure 1. Unlike some heuristic algorithms, Axe is *complete* in the sense that it will detect *any* violation of the model.

Following Gibbons [7] and Manovit [2], we assume that the address-value pair of every store in a trace is unique, i.e., the same value is never written to the same address more than once. This reduces the amount of nondeterminism in a model, because the store read by any load can be uniquely identified. The restriction is easily met by an automatic test generator, and is justified by the fact that the actual values being stored do not typically affect any interesting hardware behavior. But it does mean that our tool cannot be used for checking memory traces that arise during execution of arbitrary software applications, which are unlikely to meet this restriction.

Another technique for reducing nondeterminism is to modify the hardware to emit extra trace information such as the order in which writes reach a particular internal merge point in the memory subsystem. However, we treat the memory subsystem as a *black box*, and do not inspect or modify its internals in any way: we would like our tool to be as easy as possible to use, i.e., not requiring modifications to the system under test.

A. Syntax of memory traces

Example 1 Here is a simple Axe trace consisting of five operations running on two threads.

```
0: M[0] := 1
0: sync
0: M[1] := 1
1: M[1] == 1     @ 100 : 110
1: M[0] == 0     @ 115 :
```

The first number on each line denotes the hardware thread id; `M[`a`]` denotes a memory location with address a; operators `==` and `:=` denote loads and stores respectively; `sync` denotes a full memory barrier; the optional timestamps beginning with `@` denote the begin and end times at which the request was sent and the response received, respectively.

The textual order of operations with the same thread id is the order in which those operations were submitted to the memory subsystem by that thread. We refer to this order as the *thread-order*. No ordering is implied by the textual order of operations from different threads.

The initial value of every memory location is implicitly zero. For any load of a value other than zero, there must exist a write of that value to the same address in the trace, otherwise the trace is invalid. As explained above, we also require the address-value pair of every write to be unique.

Load operations will typically contain two timestamps, since they involve both a request and a response. Axe currently forbids response timestamps on store operations, making it clear that this information is not used by any of the supported models. All timestamps are completely optional, for a few reasons:

1) Some consistency models are unaffected by timestamps.
2) Timestamps may not be available, depending on how the traces are produced.
3) Example traces are easier to read if only the interesting or relevant timestamp information is supplied.

However, in some consistency models timestamps can affect whether or not a trace is allowed. In the above example, the timestamps indicate that the first load must have finished before the second load begins, implying that the memory subsystem could not have executed the operations out of order. In the SPARC and POWER architectures, a programmer can arrange such a dependency by having the address of the second load be dependent on the result of the first – a so-called *address dependency* [12]. Other kinds of dependency include *data dependencies* (where the value of a store is dependent on the result of a preceding load) and *control dependencies* (where an operation is control-flow dependent on the result of preceding load). These program-level dependencies become observable in the memory trace as end-time-before-begin-time dependencies.

For the SPARC models, Axe considers timestamps to be *local to each thread*, i.e., it does not use timestamps to infer ordering between operations that run on different threads.

There is no explicit support in Axe for *canceled operations*, which often arise in modern CPUs due to speculative execution or exceptions. Traces containing such operations can still be checked by simply replacing them with no-ops. There is also no support for *mixed-width* accesses at present: Axe abstracts over the width of each memory location, and hence the width may vary between traces – but *not within a trace*.

Example 2 Here is another trace, this time containing three operations, the first of which is an atomic read-modify-write.

```
0: <M[0] == 0; M[0] := 1>
1: M[0] := 2
1: M[0] == 1
```

The first line can be read as thread 0 *atomically* reads value 0 from memory location 0 and updates it to value 1. The two memory addresses in an atomic operation must be the same, otherwise the trace is invalid. In the future, it may be desirable to generalize read-modify-write (RMW) to allow any number of operations on any number of addresses, i.e., transactional memory [23].

A common way to express atomic operations in RISC instruction sets is via a pair of *load-linked* and *store-conditional* operations. At the trace level, it is straightforward to convert such a pair into a single read-modify-write:

- If the store-conditional fails, then remove it from the trace and convert the load-linked to a standard load.
- Otherwise, convert both operations to a single read-modify-write operation.

For read-modify-write operations, the response timestamp simply denotes the time at which the read-response is received.

B. Consistency models

We now introduce the supported consistency models by example; a full operational semantics for each model is available in the Axe manual [14]. Lamport's *sequential consistency* [10] is the strongest supported model; it requires that there exists a sequential interleaving of each thread's operations satisfying the trace.

Example 3 (SB) Here is a trace, known as the "store buffer" (SB) trace, that is forbidden by sequential consistency.

```
0: M[1] := 1
0: M[0] == 0
1: M[0] := 1
1: M[1] == 0
```

There are six possible interleavings of each thread's operations and none result in *both* reads returning zero. However, under TSO [11], stores may be buffered locally by a thread, allowing subsequent loads to complete before the buffered stores can be observed globally.

Example 4 (SB+syncs) Under all consistency models, the above behavior can be prevented by inserting a `sync` after the store on each thread; `sync` has the effect of flushing the store buffer of the calling thread. Such memory barriers are necessary to implement Peterson's mutual exclusion algorithm [20], for example.

Example 5 (SB+RMWs) Under TSO, another way to prevent the SB behavior is to replace each write with an atomic RMW, which has the side-effect of flushing the store buffer.

```
0: <M[1] == 0; M[1] := 1>
0: M[0] == 0
1: <M[0] == 0; M[0] := 1>
1: M[1] == 0
```

Example 6 (MP) The following "message passing" trace is forbidden under both sequential consistency and TSO.

```
0: M[0] := 1
0: M[1] := 1
1: M[1] == 1
1: M[0] == 0
```

However, under PSO [11], this is allowed: buffered stores (to different addresses) can be evicted out-of-order. Hence, the second store can be observed globally before the first.

Example 7 (MP+sync) Under PSO, the above behavior can be disallowed by inserting a `sync` between the two stores. However, MP+sync is still allowed by WMO, which permits load buffering as well as store buffering. As a result, the first load may now be buffered and overtaken by the second as they access two different addresses.

Example 8 (MP+syncs) One way to prevent the two loads from being reordered is simply to place another `sync` between them; `sync` waits for all buffered loads to complete.

Example 9 (MP+sync+dep) Another situation in which load reordering is disallowed is when a timestamp dependency forbids it. This trace is disallowed by WMO:

```
0: M[0] := 1
0: sync
0: M[1] := 1
1: M[1] == 1  @ 100 : 110
1: M[0] == 0  @ 115
```

Example 10 (LB) The MP+sync example demonstrates reordering of loads, but WMO also allows reordering of a load followed by a store. The following trace is allowed by WMO.

```
0: M[0] == 1
0: M[1] := 1
1: M[1] == 1
1: M[0] := 1
```

Example 11 (LB+syncs & LB+deps) As expected, a `sync` after each load will prevent the behavior. So too will a timestamp dependency between each load and store.

In summary: TSO allows store-load reordering; PSO additionally allows store-store reordering (when the addresses differ); WMO additionally allows load-load and load-store reorderings (when the addresses differ). It is quite easy to see how all these behaviors could arise in the presence of nonblocking L1 caches: any operation that misses in the L1 cache and is buffered may be overtaken by a subsequent operation that hits. Such behavior is important for out-of-order processors, where unnecessary dependencies between operations must be avoided.

The common feature of all these models is the existence (or illusion) of a *single shared memory*: if a write by one thread is observed by another, then it must be observable to all threads. Sometimes known as *multi-copy atomicity* or *global store atomicity*, this property is provided by hardware that implements a single-writer coherence protocol such as MESI.

C. Axiomatic definitions

We now present the axiomatic definitions for the consistency models, upon which the Axe checking algorithm is based. In these definitions, we consider a read-modify-write operation to be both a "load" and a "store".

To begin, it is helpful to distinguish between two different orderings over operations in the trace:

- *Thread Order*: for any given thread, the textual order of operations in the trace issued by that thread.
- *Memory Order*: a total order over all operations.

All valid traces under these models must satisfy the following property (**value axiom**): the value returned by a load from address a equals the value of the latest store (in memory order) from the set $Local \cup Global$ where $Local$ is the set of stores to address a that precede the load in *thread order* and $Global$ is the set of stores to address a that precede the load in *memory order*.

Depending on the model, the following **local axioms** on operations i and j from the same thread must also be satisfied.

SC If i precedes j in thread-order, then i must precede j in memory order.

TSO If i precedes j in thread-order, then i must precede j in memory order when i is a load; or i and j are stores; or i is a `sync` or j is a `sync`.

PSO If i precedes j in thread-order, then i must precede j in memory order when: i is a load; or i and j are stores *to the same address*; or i is a `sync` or j is a `sync`.

WMO If i precedes j in thread-order, then i must precede j in memory order when: i is a load and j accesses the same address; or i and j are stores to the same address; or i is a `sync` or j is a `sync`; or i is a load with end-time t_0 and j has begin-time t_1 and $t_0 < t_1$.

D. Checking algorithm

In this section, we generalize an algorithm for checking traces against the TSO model to support the SC, TSO, PSO *and* WMO models. The central data structure used by this algorithm is the *analysis graph* – in which each node denotes an operation from the trace, and each edge denotes that the source node precedes the destination node in memory order.

Simple algorithm Starting with an empty analysis graph, a simple checking algorithm is as follows.

1) Add each operation in the trace as a node to the analysis graph and add the edges implied by the local axioms defined above. (Redundant edges implied by transitivity need not be added.)
2) Apply the two edge-introduction rules shown in Figure 2 to the graph.
3) Add an edge from each read `M[`x`] == 0` to the first store `M[`x`] :=` v on each thread. This ensures that any read of zero (initial value) from address x must happen before any writes to address x.
4) Apply a standard topological sort procedure to the analysis graph with the following tweak: every time a store operation `M[`x`] :=` v is removed from the graph, add an edge from each load `M[`x`] ==` v to the next unpicked store `M[`x`] :=` w on each thread. This ensures that any read of the current value at address x must happen before any store of another value to address x.
5) If a topological sort can be found (i.e., a total order of operations exists that satisfies the memory order constraints), then the trace is valid; otherwise it is invalid.

The key inefficiency of this algorithm is the nondeterminism present in the topological sort. At any stage, there may exist several store operations that can be removed next. If a bad choice is made, the algorithm must backtrack, because an alternative choice might lead to success. (The order of stores to each address is not known in advance.)

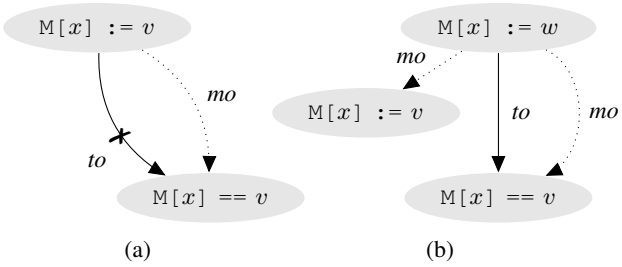

Fig. 2: Edge-introduction rules. Thread-order edges are labelled *to* and memory-order edges *mo*. In (a) the dotted edge is introduced if the solid edge *does not* exist. In (b) the dotted edges are introduced if the solid edge *does* exist and $v \neq w$.

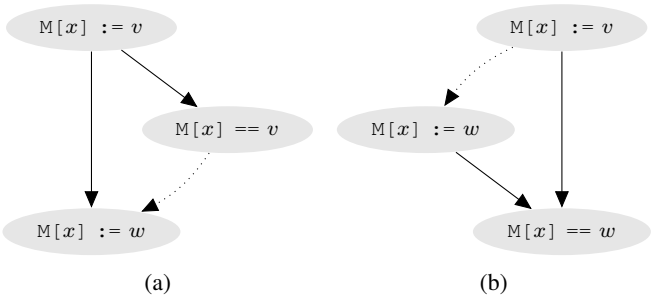

Fig. 3: Edge-inference rules proposed by Manovit [2] (our representation). All edges are memory-order edges. In each case, if the solid edges are known to exist, either directly or by transitivity, and $v \neq w$ then the dotted edge can be inferred.

Reducing nondeterminism Manovit proposes the two rules shown in Figure 3 as a way of inferring new edges in the analysis graph, greatly reducing the amount of nondeterminism in the topological sort. Notice that applying these rules can introduce edges that enable the rules to be applied again. Therefore, it is desirable to apply the rules repeatedly until a fixed-point is reached, i.e., until no new edges are inferred.

This leads to two modifications of the simple algorithm above: first, add a new step after step (2) that applies the inference rules until a fixed-point is reached; second, every time a store is removed from the graph in step (4), and new edges are added, reapply the inference rules until a fixed-point is reached.

Reducing rule-application sites Applying the inference rules at all matching sites in the analysis graph would be extremely inefficient and, fortunately, unnecessary. Manovit shows that it is sufficient to apply each rule once for each store s of the form `M[`x`] :=` v with:

- for rule 3a, node `M[`x`] :=` w bound to the earliest store to address x that succeeds s in the analysis graph;
- for rule 3b, node `M[`x`] ==` w bound to the earliest load to address x that succeeds s in the analysis graph.

While there may exist several bindings that satisfy the above

constraints (the earliest successor may not be unique in a partial order), the number of application sites to consider is greatly reduced.

Determining the earliest successors The problem now is this: starting from any store operation, how do we efficiently determine the next load and store to the same address in the analysis graph?

To answer this, we maintain two data structures. The first is the mapping $nextLoad(op, t, a)$ that gives the next load (in the analysis graph) to address a on thread t from operation op. (Since loads to the same address on a given thread are totally ordered under all models, this mapping is a function, i.e., unambiguous.) Initially, it is computed by a backward analysis, propagating the next load for each (a, t) pair backwards along the edges of the graph, in reverse topological order. At a fork point, the information at several nodes is merged by taking the minimum load in thread order for each (a, t) pair. When a new edge $i \to j$ is added to the graph, the $nextLoad$ mapping is updated by applying the same propagation method backwards from node j until no new updates are made.

The second data structure we maintain is the mapping $nextStore$, identical to $nextLoad$ but giving the next store instead of the next load. These two data structures have a number of uses:

1) The inference rules from Figure 3 can be efficiently applied. And when adding an edge, the backward-propagation method used to update the $nextLoad$ and $nextStore$ mappings will naturally visit all the nodes at which the inference rules must be reapplied.
2) The existence of a path from a store to any load or store can be determined in constant-time, avoiding the addition of redundant edges to the graph.
3) Similarly, we can be determine in constant-time whether or not the addition of an edge to the graph will lead to a cycle, allowing immediate failure detection.

Comparison to Manovit's algorithm When specializing the algorithm to the TSO model, it is possible to simplify the $nextLoad$ and $nextStore$ mappings. Instead of mapping each (op, a, t) triple to the next load or next store, it is sufficient to map each (op, t) pair. This is because all loads by the same thread are totally ordered under TSO, as are all stores by the same thread. Once the next load on some thread is determined, the next load to a particular address on that thread can be easily found by looking at the static thread order. Consequently, the size of these data structures reduces from $2 \times N \times A \times T$ for N operations, A addresses, and T threads to $2 \times N \times T$. Not only does this save space, but it makes the backward analysis faster as the amount of information being propagated is smaller. In other words, the efficiency of our checker depends on the number of different address locations used in the trace. This is not the case for Manovit's TSO-only checker.

E. Evaluation

Performance To evaluate the performance of Axe, we have generated a range of traces[2] with various numbers of memory operations ($n \in \{8K, 16K, 24K, 32K\}$), threads ($t \in \{4, 16, 32\}$), and addresses ($a \in \{4, 16, 32\}$). For each combination of parameters, we generate 16 traces, giving 576 traces in total. Figure 4 shows how the performance of the WMO checker varies with the number of operations and threads present, averaged over the number of addresses present: in practice, Axe allows rapid checking of large traces which, for a fixed number of addresses and threads, scales linearly with the number of operations.

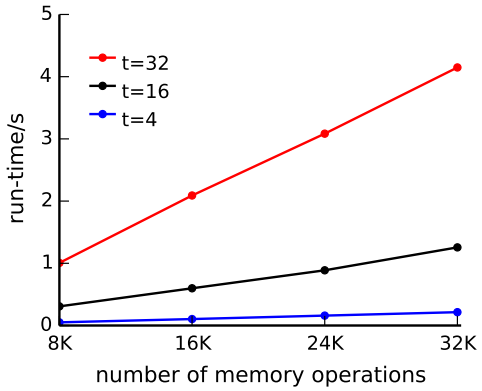

Fig. 4: Performance of the WMO checker

Correctness Axe has been tested for equivalence against an operational semantics for each model (defined in the Axe manual [14]) and also an axiomatic semantics for each model (defined in §II-C). The test traces include: (1) 199 litmus tests from the PPCMEM distribution [13]; and (2) 200K randomly-generated traces ranging from around 10 to 50 operations in size (distributed with the Axe tool [14]). Axe also gives the expected outcomes for all the traces used in our performance evaluation.

F. Shrinking traces

Given a trace that violates a model, we would like to find the smallest subset of the trace that still violates the model. This is very useful for debugging (§III). Our shrinker works by applying each of the following rewrite rules for $retry$ attempts before moving on to the next rule. Each rule is conditioned on the resulting trace still violating the model.

1) Pick an address and drop all accesses to that address.
2) Drop a random subset ($n\%$) of loads.
3) Drop a random subset ($n\%$) of stores which write a value that is never read.
4) Repeat (3) but for read-modify-write operations.

After that, in reverse trace order, it tries to drop each operation in turn; this is repeated until a fixed-point is reached. For suitable choices of $retry$ and n, the shrinker is both effective and fast, typically yielding fewer than ten operations and taking between a second and a minute for traces between 1K and 32K elements respectively.

III. Case study: Rocket Chip

Rocket Chip is an open-source system-on-chip generator developed at UC Berkeley including support for multiple pro-

[2]Using a model cache implementation with load and store buffering, out-of-order eviction, out-of-order responses, prefetching and invalidation-based coherence. These traces are available at http://dx.doi.org/10.17863/CAM.794

```
1: load-req      0x0000000008 #0 @64
1: store-req  5  0x0000100008 #1 @65
1: store-req  7  0x0000000010 #2 @66
0: store-req  2  0x0000000008 #0 @303
0: load-req      0x0000000008 #1 @304
0: store-req  6  0x0000100008 #2 @305
1: resp       0               #0 @96
0: resp       0               #0 @350
0: resp       2               #1 @351
0: load-req      0x0000000010 #3 @353
1: resp       0               #1 @149
1: load-req      0x0000000108 #3 @152
1: resp       0               #3 @184
0: resp       5               #2 @422
0: resp       0               #3 @424
1: resp       0               #2 @226
```

Fig. 5: A sample trace generated using our extensions to Rocket Chip's GroundTest framework: the first number on each line of the trace is the *thread-id*; `#`n denotes a *request-id* n; `@`t denotes a *time* t in clock cycles; hex numbers denote *addresses*; remaining decimal numbers denote *data values* being loaded or stored. This trace contains only *loads*, *stores* and *responses*, but we also support generation of *LR/SC* pairs, *atomic operations*, and *fences*. Notice that the timestamps are not monotonically increasing: in simulation, the Rocket Chip tiles are brought out of reset sequentially.

cessor cores and a cache-coherent shared-memory subsystem. Available cores include implementations of the RISC-V ISA: the in-order Rocket, the out-of-order BOOM, and the Z-scale microcontroller – and (pending release) the Hwacha vector-thread accelerator. Having been taped out 11 times between 2011 and 2015, Rocket Chip is fairly mature – but faces constant change through extensions, redesigns, and refactorings. Rocket Chip is written using the Chisel HDL [16].

The Rocket Chip developers have already recognized the importance of making HDL-level test benches for the memory subsystem: *"In order to test behaviors in our memory hierarchy which are not easy or efficient to test in software, we have designed a set of test circuits called GroundTest"* [9]. GroundTest plugs into the socket given to CPU tiles and generates various kinds of memory traffic directly to the memory subsystem, either via the L1 caches, or directly to the L2, or via DMA.

Rocket Chip is highly parameterized, including the choice of coherence protocol – which by default is MESI, at the time of writing. Since MESI guarantees at most one writer to a cache line at any time, it gives the illusion of a single shared memory – despite the reality of multiple local caches – and is thus expected to conform to one of the SPARC consistency models.

Extending GroundTest We developed a *trace generator* that plugs into the GroundTest framework. Given a random seed, it generates random memory requests from each tile, and emits a trace of events. To illustrate, Figure 5 shows an example of a generated trace.

The number of tiles, requests, and addresses used when generating a trace can all be controlled using compile-time parameters. Ideally though, the number of requests and addresses would be taken as simulation-time parameters, allowing the top-level testing script to gradually increase the sizes of traces in the hope of finding smaller failures first. Unfortunately, Chisel does not yet support a convenient way to read from external sources (such as files or environment variables) during simulation.

Converting traces to Axe format We made a simple script to convert traces emitted by the trace generator into Axe format. For example, given the sample trace from Figure 5, this conversion script yields:

```
# &M[2] == 0x0000000010
# &M[0] == 0x0000000008
# &M[3] == 0x0000000108
# &M[1] == 0x0000100008
1: M[0] == 0 @ 64:96
1: M[1] := 5 @ 65:
1: M[2] := 7 @ 66:
0: M[0] := 2 @ 303:
0: M[0] == 2 @ 304:351
0: M[1] := 6 @ 305:
0: M[2] == 0 @ 353:424
1: M[3] == 0 @ 152:184
```

Notice that lines beginning with `#` are treated as comments by Axe: we use these comments to record the mapping between physical addresses and addresses used by Axe.

Testing against the SC model We made a script that repeatedly: (1) generates a trace with a random seed; (2) converts the trace to Axe format; and (3) checks the trace against the chosen consistency model. Running this script, we found a 260-element trace that fails to satisfy sequential consistency. Passing this through our shrinking procedure (§II-F), we get:

```
1: M[1] := 185 @ 1921:
1: M[0] := 193 @ 1966:
0: M[0] == 193 @ 2207:2245
0: M[1] := 204 @ 2208:
0: M[1] == 185 @ 2209:2269
```

Similar to the MP example, this trace can be explained either by thread 1's stores being performed out-of-order (PSO) or thread 0's loads being performed out-of-order (WMO).

Testing against the PSO model We also found a 261-element trace that violates PSO, which after shrinking is:

```
0: M[2] == 137 @ 1825:1948
0: M[0] := 154 @ 1886:
1: M[0] == 154 @ 1689:1725
1: M[2] := 137 @ 1690:
```

Similar to the LB example, this trace can be explained by the load and store on thread 0 (or 1) being reordered (WMO).

Coherence bug We observed that a large number of traces satisfy the WMO model, but eventually we hit a 260-element counterexample – which after shrinking is:

```
0: M[2] := 46 @ 497:
1: M[2] == 46 @ 280:513
1: M[2] := 61 @ 729:
1: M[2] == 46 @ 854:979
```

Note that the write of `M[2] := 46` by core 0 is the only write of 46 in the entire trace (the trace generator ensures that all write values are unique). Also, the initial value of each location is 0. Therefore, the write `M[2] := 61` by core 1 has seemingly been dropped. This is a coherence violation and undesirable: if the write of 46 to `M[2]` is interpreted as "core 1, a message is available", then core 1 might end up receiving two messages as it effectively sees the write twice. (It sees 46 once on line 2, then it clears that value with a store on line 3, and finally it sees 46 again on line 4). We reported this issue to the Rocket Chip developers, who identified a race condition in the coherence protocol and fixed it within a few days.

Livelock bug For the above testing we enabled only loads and stores in the trace generator. When we enabled generation of LR/SC pairs, we found a lock-up issue in which a store-conditional would never return under some circumstances. We reported this to the Rocket Chip developers, who diagnosed the problem as a livelock issue in the coherence protocol.

Store-conditional bug With the livelock issue fixed, we found a 228-element counterexample to WMO. After shrinking it is:

```
1: M[3] := 31 @ 340:
0: { M[3] == 31;  M[3] := 178} @ 745:812
0: { M[3] == 178; M[3] := 198} @ 926:955
1: { M[3] == 178; M[3] := 59 } @ 759:761
```

Notice that the read-modify-write by thread 1 atomically changes `M[0]` from 178 to 59. Furthermore, the second read-modify-write by thread 0 atomically changes `M[0]` from 178 to 198. Of course, if these operations really were atomic, this behavior would be impossible. After investigating the raw trace emitted by the generator, we noticed this issue arises when a store-conditional is issued before a load-reserve response is received. We reported this issue to the Rocket Chip developers, who identified it as a bug in which a cache line is not marked as dirty when it should be.

Testing against the WMO model At the time of writing (with loads, stores, LR/SC pairs, atomics, and fences all being generated), Rocket Chip satisfies the WMO model on thousands of large traces, each comprising 64K operations, 16 addresses, and 8 threads.

Liveness A key limitation of the above specification-based testing approach is that it does not check for liveness, e.g., that a store-conditional operation actually succeeds when it should. In response, we added a mode to the trace generator in which it will generate only LR/SC pairs that are expected to succeed. This is possible in Rocket Chip because of the way it implements LR/SC: the L1 cache will hold on to a cache line for a maximum of n cycles after an LR response. Thus, provided the LR and SC are within n cycles of each other, the SC should succeed. In this mode, we observed an LR/SC success rate of 94%. The 6% of failures remain unexplained and we plan to explore this in future work.

IV. Comparisons with related work

A. Litmus testing

Litmus testing is a method of determining whether or not specific memory behaviors are observable in a multiprocessor implementation [17], [18]. Behaviors are captured by litmus tests – small concurrent program-fragments with pre- and post-conditions. To a first approximation, a litmus testing tool works by repeatedly: (1) establishing the test's pre-condition; (2) synchronizing all threads; (3) running the test; and (4) recording the value of the post condition. Slight variations are introduced on each iteration – for example by changing the addresses of the shared variables used, by inserting random delays, or by simply relying on random perturbations due to context switching and other OS activities.

```
{ x=0; 0:r2=x; 1:r2=x; }
   P0            | P1 ;
   ll  r1, 0(r2) | ll  r1, 0(r2) ;
   add r1, r1, 1 | add r1, r1, 1 ;
   sc  r1, 0(r2) | sc  r1, 0(r2) ;
exists (0:r1=0 /\ 1:r1=0)
```

Fig. 6: A litmus test for MIPS in which two threads attempt to concurrently increment a shared variable `x` using load-linked and store-conditional operations. The test looks for the case where *both* store-conditionals fail – a potential liveness bug.

In our efforts to verify the memory subsystem of the BERI multiprocessor [15], [19], we have found litmus testing to be complementary to our Axe-based approach, which does not cover liveness properties. For example, the litmus test shown in Figure 6 caught a serious liveness bug in BERI to which Axe was oblivious. Unlike in Rocket Chip, it is not possible in BERI to capture static conditions under which a store conditional is expected to succeed: concurrent LL/SC accesses to the same address are resolved by a race, and whoever wins invalidates the others. But regardless of who wins, there should exist a winner. The litmus test was able to disprove this by showing a case where all store-conditionals fail. This was due to a bug in which even a failing store-conditional would invalidate the load-linked reservations of other threads.

Litmus testing, as described in [18], is a whole-system approach, covering the processor pipeline, the memory subsystem, and even the compiler. This strength is also a weakness when it comes to modular reasoning and debugging. For example, our test framework for BERI – which employs Axe alongside techniques for finding simple failures [19] – can find a counterexample to sequential consistency containing just five operations. Litmus testing can require hundreds of iterations, with hundreds of memory accesses per iteration, to expose the same behavior. Using the smaller counterexample, it is fat easier to manually trace through the internal hardware state transitions to understand *why* the behavior is occurring.

Another problem with a whole-system approach is that a largely complete and largely working SoC is required before testing can be attempted. The complex software mechanism used to synchronize all threads at the beginning of each litmus test iteration is, on its own, a very demanding test. In contrast, our approach allows incremental development, starting out with plain load and store requests, and later moving to fences and atomics – all without the need for a CPU pipeline, a compiler, or an OS.

Finally, litmus tests consider specific – not arbitrary – sequences of memory operations, and we developed Axe to support full specification-based testing.

B. Intel's checker

We are not the first to generalize TSOtool's checking algorithm to a wider range of consistency models. Intel has incorporated a variant of the algorithm into their MP RIT (multiprocessor random instruction test) framework [8], with support for any consistency model that provides global store atomicity. This means that Intel's algorithm is more general than ours. However, this extra generality comes at a cost, and its benefit is not clear cut.

Cost Intel's algorithm represents the analysis graph as an adjacency matrix and maintains the full transitive closure. Axe exploits the fact that, in WMO, loads to the same address on each thread are totally ordered, as are stores to the same address. This means we don't need to track all successors for each node; we need only track the nearest successor of each node for each (address,thread) pair. The extra cost of Intel's algorithm is apparent in their performance graph: despite parallelizing the checker over multiple cores, a polynomial growth in execution time is observed for traces up to just 8K operations for a fixed 8 threads.

Benefit The benefit of the increased generality over WMO is unclear. Both WMO and Intel's checker require global store atomicity. WMO additionally requires sequential-consistency-per-location, but this property is provided by almost all CPUs [24].

Unlike Axe, Intel's checker is *incomplete*, i.e., there are some model violations that the tool will inherently miss. And as with TSOtool, Intel's checker is not publicly available.

V. Conclusions

We have generalized a state-of-the-art TSO conformance-checking algorithm to support a wider range of consistency models increasingly being found in modern hardware. Although the generalized algorithm has worse time and space complexity – now dependent on the number of distinct memory locations that are accessed – it still performs very well in practice. Using it, we have been able to test the memory subsystem of Berkeley's Rocket Chip, which supports load buffering and out-of-order responses and is therefore intentionally more relaxed than TSO. This testing has uncovered a number of serious memory consistency bugs that have been reported to the Rocket Chip developers in a clear and concise manner using our trace shrinking procedure. In contrast to whole-system approaches, we have focused on testing the memory subsystem as a reusable component that can be understood in isolation and verified incrementally. This not only permits testing at much earlier stage in the development process, but also leads to simpler failure cases. Axe is now part of the standard test infrastructure for both the BERI and Rocket Chip open-source processors.

Acknowledgements Many thanks to Henry Cook, Howard Mao, Peter Neumann, Peter Sewell, Andrew Waterman, Jon Woodruff, and the anonymous reviewers. This work was supported by DARPA/AFRL contracts FA8750-10-C-0237 (CTSRD) and FA8750-11-C-0249 (MRC2), and EPSRC grant EP/K008528/1 (REMS). The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

Open access Research data supporting this paper can be obtained from `http://dx.doi.org/10.17863/CAM.794`. This includes all the sample traces used to test and evaluate the performance of Axe, and a snapshot of the Axe source code taken in July 2016. However, the latest version of the Axe source code should always be obtained from `https://github.com/CTSRD-CHERI/axe`.

References

[1] S. Adve and K. Gharachorloo. *Shared Memory Consistency Models: A Tutorial*, Computer Journal, volume 29, number 12, pp. 66–76, 1996.

[2] C. Manovit. *Testing memory consistency of shared-memory multiprocessors*, PhD thesis, Stanford University, 2006.

[3] S. Hangal, D. Vahia, C. Manovit, and JY. J. Lu, *TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model*, in ISCA 2004, pp. 114.

[4] C. Manovit and S. Hangal, *Efficient algorithms for verifying memory consistency*, in SPAA 2005, pp. 245–252.

[5] C. Manovit and S. Hangal, *Completely verifying memory consistency of test program executions*, in HPCA 2006, pp. 166–175.

[6] *Homepage of TSOTool*, a program for verifying memory systems using the memory consistency model, http://xenon.stanford.edu/~hangal/tsotool.html.

[7] P. B. Gibbons and E. Korach. *On testing cache-coherent shared memories*, in SPAA 1994, pp. 177-188.

[8] A. Roy, S. Zeisset, C. J. Fleckenstein, J. C. Huang, *Fast and Generalized Polynomial Time Memory Consistency Verification*, CAV 2006, pp. 503.

[9] K. Asanovic et al., *The Rocket Chip Generator*, Technical Report UCB/EECS-2016-17, University of California, Berkeley, 2016.

[10] L. Lamport. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, volume 28, number 9, pp. 690–691, 1979.

[11] D. L. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*, 2003.

[12] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. *Understanding POWER Multiprocessors*, PLDI 2011, pp. 175–186.

[13] *Homepage of PPCMEM/ARMMEM*, a tool for exploring the POWER and ARM memory models, https://www.cl.cam.ac.uk/~pes20/ppcmem/.

[14] M. Naylor, S. Moore, and A Mujumdar, *Axe Manual Version 1.4*, https://github.com/CTSRD-CHERI/axe.

[15] *Homepage of the BERI processor (Bluespec Enhanced RISC Instructions)*, http://bericpu.org.

[16] J. Bachrach et al. *Chisel: Constructing Hardware in a Scala Embedded Language*, DAC 2012, pp. 1216–1225.

[17] S. Sarkar, P. Sewell, F.Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M.O. Myreen, and J. Alglave, *The Semantics of x86-CC Multiprocessor Machine Code*, in POPL 2009, pp. 379–391.

[18] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, *Litmus: Running Tests Against Hardware*, in TACAS 2011, pp. 41–44.

[19] M. Naylor and S. W. Moore, *A Generic Synthesisable Test Bench*, in MEMOCODE 2015, pp. 128–137.

[20] G. L. Peterson, *Myths About the Mutual Exclusion Problem*, Information Processing Letters, vol. 12, no. 3, pp. 115-116, 1981.

[21] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*, EECS Department, University of California, Berkeley, May 2014.

[22] W. M. Collier. Reasoning about parallel architectures. Prentice-Hall, Inc., 1992.

[23] C. Manovit, S. Hangal, H. Chafi, A. McDonald, and C. Kozyrakis, K. Olukotun, *Testing Implementations of Transactional Memory*, in PACT 2006, pp. 134–143.

[24] J. Alglave, L. Maranget, and M. Tautschnig, *Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory*, in ACM TOPLAS, volume 36, number 2, 2014.